Clase 3 — 28.10.25

#IntelliJIDEA #JAVA

👨‍🏫 Profesor: Álvaro García Gutierrez
📘 Acceso a Datos
🗓️ Clase 3 — 28/10/2025
🎯 Tema: Ficheros Binarios en Java

1️⃣ Introducción: ¿Qué son los ficheros binarios?

Los **ficheros binarios** almacenan la información en **formato binario (bits)**, tal como la interpreta la máquina.
No están pensados para que un humano los lea, a diferencia de un fichero de texto.

🔹 Diferencias clave respecto a los ficheros de texto

- **No** usan saltos de línea ni separadores.
- Los datos se guardan **tal cual están en memoria**.
- Son más **eficientes**, compactos y rápidos de procesar.
- **No se pueden abrir y leer directamente** en un editor de texto sin ver símbolos extraños.

Esto es especialmente útil cuando se trabaja con:

- Registros estructurados.
- Lecturas y escrituras de alto rendimiento.
- Acceso aleatorio a datos.

2️⃣ Tamaño de los tipos de datos en binario

Comprender este apartado es **fundamental**, porque en ficheros binarios cada dato ocupa un número exacto de bytes.
El profesor ha señalado especialmente estos tamaños con los que trabajaremos.

Tipo de dato	Tamaño (bytes)	Descripción
`byte`	1	Entero pequeño (-128 a 127)
`short`	2	Entero corto
`int`	4	Entero estándar
`long`	8	Entero largo
`float`	4	Número real simple
`double`	8	Número real doble precisión
`char`	2	Carácter Unicode (⚠ cada carácter = **2 bytes**)
`boolean`	1	Verdadero/falso (0 o 1 aprox.)

🧩 Ejemplo

Un registro con:

- `int id` → 4 bytes
- `double salario` → 8 bytes
- `char[10] nombre` → 20 bytes

Total del registro: 32 bytes

Esto es clave para el **acceso aleatorio**:

- Si cada registro ocupa 32 bytes,
- entonces el registro *n* empieza en la posición:
  `n × 32`.

🔍 **Comentario del profesor:**

> Cada carácter ocupa 2 bytes. Por eso un `char[10]` son 20 bytes y no 10.
> En binario necesitamos saber el tamaño exacto para poder saltar registros sin leer los anteriores.

3️⃣ Clases principales para trabajar con binarios en Java

En Java existen varias clases para manejar ficheros binarios.

Cada una tiene un propósito muy concreto y es importante saber **cuándo usar cada una**.

Las tres clases clave son:

- **FileOutputStream**
- **FileInputStream**
- **RandomAccessFile**

Vamos a verlas en profundidad.

3.1 FileOutputStream

¿Qué es?

Es una clase pensada para **escribir bytes** en un archivo binario de manera **secuencial** (es decir, desde el principio hasta el final, sin saltos).

Internamente funciona así:

```
[buffer RAM] → fos.write() → [fichero.bin]
```

El flujo de salida escribe **byte a byte** en el archivo.

¿Cuándo se usa?

- Para escribir datos "sin formato" (raw bytes).
- Para guardar imágenes, audio, vídeos…
- Para escribir texto como bytes (codificación por defecto o la que indiques).
- Para volcar arrays de bytes, buffers, streams de red…

Código

```
import java.io.FileOutputStream;
import java.io.IOException;

public class EjemploFileOutputStream {
    public static void main(String[] args) {
        String texto = "Hola binario";
        try (FileOutputStream fos = new            FileOutputStream("ejemplo.dat")) {
            fos.write(texto.getBytes());               // convierte texto a bytes y
los escribe
            System.out.println("Fichero                binario creado con
FileOutputStream");
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

Explicación detallada

```
import java.io.FileOutputStream;
import java.io.IOException;
```

- Importas las clases necesarias:
  - `FileOutputStream`: para escribir bytes en un fichero.
  - `IOException`: excepción que se lanza cuando falla E/S (archivo no existe, permisos, etc.).

```
public class EjemploFileOutputStream {
```

- Declaras la clase pública con ese nombre. Debe llamarse igual que el fichero `.java`.

```
public static void main(String[] args) {
```

- Punto de entrada del programa.
- `static` para poder ejecutarse sin instanciar la clase.

```
        String texto = "Hola binario";
```

- Creas una cadena en memoria; todavía no hay nada en el disco.
- En RAM es una secuencia de `char` (2 bytes por carácter).

```
        try (FileOutputStream fos = new            FileOutputStream("ejemplo.dat")) {
```

- Abres un bloque **try-with-resources**:
  - `new FileOutputStream("ejemplo.dat")`:
    - Si el fichero no existe → lo crea.
    - Si existe → lo sobrescribe desde el principio.
  - La variable `fos` es el flujo de salida hacia ese archivo.

- Al usar `try (...)` Java cerrará automáticamente `fos` al salir del bloque (aunque haya error).

```
            fos.write(texto.getBytes());
```

- `texto.getBytes()`:
  - Convierte la cadena `"Hola binario"` a un **array de bytes** usando una codificación (normalmente UTF-8).
  - Ej.: `['H','o','l','a',' ','b','i',...]` → `[48 6F 6C 61 20 62 69 ...]` en bytes.
- `fos.write(...)`:
  - Escribe **todos esos bytes** en el fichero `ejemplo.dat` desde el principio.

```
            System.out.println("Fichero                binario creado con
FileOutputStream");
```

- Mensaje informativo: no toca el fichero; solo informa por consola.

```
        } catch (IOException e) {
            e.printStackTrace();
        }
```

- Si algo falla en el `new FileOutputStream` o en el `write`:
  - Se captura la `IOException`.
  - `printStackTrace()` muestra la traza de error para depuración.

```
    }
}
```

- Cierre de `main` y de la clase.
- Al salir del bloque `try`, `fos.close()` se llama automáticamente.

Puntos clave

1. ¿Por qué `getBytes()`?

Porque FileOutputStream **NO sabe nada de caracteres**.

Él solo sabe escribir **bytes**, así que hay que convertir antes:

```
String → codificación → bytes
```

Por defecto usa la codificación del sistema (UTF-8 normalmente), pero se recomienda:

```
texto.getBytes(StandardCharsets.UTF_8);
```

---

2. ¿Sobrescribe o añade?

- Sobrescribe por defecto.
- Para **añadir**, se debe usar:

```
new FileOutputStream("archivo.dat", true)
```

---

3. Errores comunes

- Intentar escribir texto sin convertirlo a bytes → ❌ error.
- Abrir el fichero sin cerrar → provoca corrupción.
  (Try-with-resources lo soluciona.)
- Mezclar caracteres con binario sin controlar la codificación.

3.2 FileInputStream

¿Qué es?

Clase para **leer bytes** de un fichero binario, también **secuencialmente**.

Funciona así:

```
[fichero.bin] → fis.read() → [RAM]
```

Lee **de principio a fin**, sin capacidad de saltar posiciones (aunque puedes hacerlo manualmente usando `skip()`).

Código

```
import java.io.FileInputStream;
import java.io.IOException;

public class EjemploFileInputStream {
    public static void main(String[] args) {
        try (FileInputStream fis = new             FileInputStream("ejemplo.dat")) {
            int byteLeido;
            while ((byteLeido =                        fis.read()) != -1) {
                System.out.print((char)                    byteLeido); // Convertimos
a char para ver el texto
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

Explicación detallada

```
import java.io.FileInputStream;
import java.io.IOException;
```

- `FileInputStream`: flujo de **entrada** de bytes desde un fichero.
- `IOException`: para capturar errores de E/S.

```
public class EjemploFileInputStream {
    public static void main(String[] args) {
```

- Clase y método principal, como antes.

```
        try (FileInputStream fis = new            FileInputStream("ejemplo.dat")) {
```

- Abres el fichero `ejemplo.dat` para lectura binaria.
- Si no existe → lanza `FileNotFoundException` (subtipo de `IOException`).
- `fis` es el flujo que va **del fichero a tu programa**.
- try-with-resources → se cerrará solo al terminar.

```
            int byteLeido;
```

- Declaras una variable `int` para almacenar cada byte leído.
- Aunque se lea 1 byte, `read()` lo devuelve en un `int` (0–255) para poder usar `-1` como señal de fin.

```
            while ((byteLeido =                          fis.read()) != -1) {
```

- Bucle de lectura clásico:
  - `fis.read()` lee **un byte** del fichero.
  - Si hay datos → devuelve un valor entre 0 y 255.
  - Si se ha llegado al final → devuelve `-1`.
- La asignación dentro del while:
  - Asigna el resultado a `byteLeido`.
  - Compara si es distinto de `-1` para seguir.

```
                System.out.print((char)                    byteLeido);
```

- Convierte el número leído (por ejemplo `72`) a un `char` (`'H'`).
- Lo imprime sin salto de línea (`print`).
- Esto solo da buen resultado si el fichero contiene texto en una codificación compatible con ASCII/UTF-8 simple.

```
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
```

- Cierre del while.
- El catch captura cualquier `IOException` de apertura o lectura.
- El flujo `fis` se cierra automáticamente al salir del `try`.

Puntos clave

1. ¿Qué devuelve `read()`?

- **Un int de 0 a 255** → representa un byte leído.
- **Devuelve -1** cuando llega al final.

Por eso el while típico es:

```
while ((b = fis.read()) != -1) { ... }
```

---

2. Convertir a char

En el ejemplo convierten:

```
(char) byteLeido
```

Esto obras SOLO si el archivo contiene caracteres ASCII (0–127).

Si el fichero contiene texto en UTF-8 real, puede romperse porque UTF-8 usa bytes multibyte.

---

3. Leer varios bytes de golpe

FileInputStream permite mejorar rendimiento:

```
byte[] buffer = new byte[1024];
int n = fis.read(buffer);
```

---

4. Errores típicos

- Suponer que 1 byte = 1 carácter (no es cierto en Unicode).
- Leer carácter por carácter → lento.
- Olvidar cerrar → lock en Windows.

3.3 RandomAccessFile

¿Qué es?

`RandomAccessFile` es una clase **única** dentro de la API estándar de Java.

A diferencia de la mayoría de clases de E/S (Input/Output Streams) que solo permiten recorrer el fichero **de principio a fin**, esta clase se comporta como si el fichero fuera un **array gigante de bytes** sobre el que puedes saltar libremente.

Es, en esencia, un **puntero de archivo manipulable**.

Permite:

- Leer y escribir **simultáneamente** (en el mismo objeto).
- Utilizar operaciones de movimiento del puntero interna.
- Acceder directamente a cualquier posición del archivo **sin leer lo anterior**.

Una representación sería:

```
[0] [1] [2] [3] [4] ... [1024] ... [24576]
 ↑
```

```
 puntero inicial (posición 0)

seek(1024)
                ↑
                puntero se mueve directamente al byte 1024
```

¿Qué lo hace tan importante?

- Permite **acceso aleatorio real**: leer o modificar un dato concreto sin procesar todo el fichero.
- Está pensado para **registros de tamaño fijo**, como estructuras de C, bases de datos planas, indexación o ficheros de configuración binarios.
- Evita volver a generar el archivo completo cuando solo hay que actualizar un valor.

En la práctica se usa para simular ficheros tipo:

- DBF
- ISAM (Indexed Sequential Access Method)
- Archivos de juego (slots, inventarios, datos de personaje)
- Listas de empleados con estructura fija
- Índices de búsqueda binaria dentro del propio fichero

¿Por qué no existe nada similar en FileInputStream o FileOutputStream?

Porque:

- **FIleInputStream** solo puede leer **hacia adelante**.
- **FileOutputStream** solo escribe **hacia adelante**.
- No hay noción de puntero manipulable.

En cambio, RandomAccessFile incorpora un puntero de lectura/escritura integrado:

```
+------------------------------+
|   RandomAccessFile object    |
|  - puntero interno           |
|  - modo r/rw                 |
+------------------------------+
                |
                ▼
+-----------------------------------------------+
|              Fichero en disco                 |
|  byte0 byte1 byte2 ... byteN                  |
+-----------------------------------------------+
```

¿Cómo funciona internamente?

`RandomAccessFile` mantiene un `file pointer`, que es un número:

- Empezando en **0**
- Que se actualiza con cada lectura/escritura
- Y que puedes posicionar manualmente mediante `seek()`

Ejemplo conceptual:

```
raf.writeInt(5);   // escribe 4 bytes → puntero pasa de 0 a 4
raf.writeInt(7);   // puntero pasa de 4 a 8
raf.seek(0);       // puntero vuelve al byte 0
```

Explicación del mecanismo fundamental

Cuando haces:

```
seek(n)
```

la clase:

1. Calcula la posición física deseada.
2. Le pide al sistema operativo que mueva el puntero del descriptor de archivo.
3. A partir de ese momento, **cualquier operación de lectura o escritura será en esa posición**.

Esto se parece a manipular una cinta magnética, pero en disco:

```
          writeData()
               |
               ▼
DISCO   [----|------|---------------|---------]
              ↑
           seek(5)
```

Aplicaciones REALES

- Sistemas donde cada registro tiene un tamaño fijo, por ejemplo 36 bytes.
  Entonces para acceder al registro número `i`:

```
seek(i * 36)
```

- Lectura directa tipo “array de estructuras”
- Editor hexadecimal
- Lectura de cabeceras en archivos multimedia
- Actualización de valores concretos sin reescribir todo
- Algoritmos de indexación (B+ trees implementados en fichero)

Tabla

Clase	Propósito	Acceso	Ideal para…
FileInputStream	Leer bytes	Secuencial	Decodificar binarios, leer imágenes
FileOutputStream	Escribir bytes	Secuencial	Generar binarios, volcados de datos
RandomAccessFile	Leer/escribir datos primitivos	Aleatorio	Bases de datos simples, registros, índices

En resumen:
RandomAccessFile = FileInputStream + FileOutputStream + seek()

Modos de apertura

`"r"`

- Solo lectura.
- No se permite escribir.
- Fallará si el fichero no existe.

`"rw"`

- Lectura + escritura.
- Crea el fichero si no existe.
- Permite modificar bytes ya escritos.

No existe `"w"` ni `"a"` como en FileWriter.
Para añadir contenido debes usar:

```
raf.seek(raf.length());
```

Métodos

.setLength(n) en StringBuffer

Esta parte controla **la longitud exacta** que tendrá la cadena cuando se escriba en el fichero binario.
El objetivo es que **siempre ocupe el mismo número de caracteres**, independientemente del nombre real.

¿Qué hace realmente?

- Si la cadena **tiene menos** caracteres que `n` → **rellena** con el carácter nulo `\u0000`.
- Si la cadena **tiene más** caracteres que `n` → **recorta** la cadena.
- Así se garantiza que **cada registro ocupa los mismos bytes**.

Ejemplo:

```
StringBuffer sb = new StringBuffer(nombre);
sb.setLength(10);
raf.writeChars(sb.toString());
```

- Nombre real: `"Ana"` (3 chars)
- Después del `setLength(10)` → `"Ana\u0000\u0000\u0000\u0000\u0000\u0000\u0000"`

Como **cada char ocupa 2 bytes**, se escriben **20 bytes fijos**.

Visualmente:

```
'A' 'n' 'a' '\u0000' '\u0000' '\u0000' '\u0000' '\u0000' '\u0000' '\u0000'
  ↑     ↑       ↑       nen de relleno
```

Esto es esencial para acceso aleatorio.

`writeChars()` — ¿Qué hace exactamente?

`writeChars(String s)` escribe **cada carácter** usando **2 bytes** en Unicode UTF-16.

Si la cadena (tras el `setLength`) tiene 10 caracteres → 20 bytes escritos.

Ejemplo:

```
"Ana\u0000\u0000\u0000\u0000\u0000\u0000\u0000"
```

→ Se guarda como:

```
00 41 00 6E 00 61 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

(Dos bytes por carácter)

Lectura correspondiente

Para reconstruir esos 10 caracteres:

```
char[] nombre = new char[10];
for (int i = 0; i < 10; i++) {
    nombre[i] = raf.readChar();   // lee 2 bytes por char
}
String nombreStr = new String(nombre).trim();
```

¿Qué se obtiene al leer?

Si el nombre era "Ana":

```
['A','n','a','\u0000','\u0000','\u0000','\u0000','\u0000','\u0000','\u0000']
```

¿Qué hace exactamente `.trim()`?

👇

`.trim()` elimina:

- **Espacios en blanco** al principio y al final (`' '`)
- **Carácter nulo** (`'\u0000'`)
- **Saltos de línea** (no aplica aquí)
- **Tabulaciones** (tampoco relevantes aquí)

Es CRÍTICO en este contexto porque:

1. **Los 7 caracteres nulos de relleno** al final deben desaparecer.
2. Si usáramos `String.valueOf(nombre)` sin `.trim()` obtendríamos `"Ana"` .

Por eso el PDF muestra:

```
String nombreStr = new String(nombre).trim();
```

Resultado final:

```
"Ana"
```

¿Por qué no se usa `.strip()`?

`.strip()` elimina espacios y caracteres Unicode definidos como “whitespace”,
pero **no elimina** `\u0000` (carácter nulo).

`.trim()` sí elimina `\u0000`.

Por eso en ficheros binarios solo sirve **.trim().

¿Qué pasa si no quitamos los caracteres nulos?

Si imprimes la cadena directamente → verás basura o espacios raros.

Si la usas junto a un formato fijo:

```
System.out.printf("Nombre: %-10s", nombreStr);
```

Sin `.trim()` se rompería el formato.

Ejemplo

Escritura:

```
String nombre = "Eva";

StringBuffer sb = new StringBuffer(nombre);
sb.setLength(10);
raf.writeChars(sb.toString());
```

Datos reales escritos:

```
E  v  a  \u0000  \u0000  \u0000  \u0000  \u0000  \u0000  \u0000
```

Bytes:

```
00 45 00 76 00 61 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

Lectura:

```
char[] buffer = new char[10];
for (int i = 0; i < 10; i++) {
    buffer[i] = raf.readChar();
}
String nombre = new String(buffer).trim();
```

Antes del trim:

```
"E", "v", "a", "\u0000", "\u0000", "\u0000", "\u0000", "\u0000", "\u0000", "\u0000"
```

Después del trim:

```
"Eva"
```

Perfecto.

Resumen ampliado para exámenes

- `.setLength(n)` fija un tamaño exacto de caracteres.
- Los caracteres sobrantes se rellenan con `\u0000`.
- `writeChars()` escribe cada char en **2 bytes**.
- Para leer, se usa un `char[]` del mismo tamaño.
- `.trim()` elimina los caracteres nulos y espacios.

Si quieres, ahora ampliamos también **la sección de `String.format()`**, **los ejemplos de longitud fija**, o los ejercicios de empleados. Sólo dime cuál seguimos.

Métodos de escritura

`RandomAccessFile` escribe datos **primitivos** en binario, no en texto.

Método	Bytes que escribe	Representación
`write(int)`	1 byte	Valor bajo del int
`writeBoolean()`	1 byte	0 o 1
`writeByte()`	1 byte	Byte literal
`writeChar()`	2 bytes	UTF-16
`writeShort()`	2 bytes	Big-endian
`writeInt()`	4 bytes	Big-endian
`writeLong()`	8 bytes	Big-endian
`writeFloat()`	4 bytes	IEEE-754
`writeDouble()`	8 bytes	IEEE-754
`writeUTF()`	2 bytes longitud + datos UTF-8	

Ejemplo para escribir un registro fijo:

```
raf.writeInt(1);          // 4 bytes
raf.writeChars("Ana");    // 6 bytes (3 chars × 2 bytes)
raf.writeInt(30);         // 4 bytes
```

Métodos de lectura

Los métodos complementan la escritura:

- `readBoolean()`
- `readByte()`
- `readChar()`
- `readInt()`

- `readDouble()`
- `readUTF()`

Ejemplo:

```
int id = raf.readInt();
char letra = raf.readChar();
double sueldo = raf.readDouble();
```

Cada llamada:

1. Lee el número exacto de bytes.
2. Avanza el puntero.
3. Devuelve el valor.

COMPARATIVA FINAL

Tarea	Clase ideal	Motivo
Guardar datos binarios secuenciales	FileOutputStream	Sencillo, rápido
Leer bytes en orden	FileInputStream	Modelo de flujo
Saltar entre posiciones	RandomAccessFile	El único que puede
Registros de tamaño fijo	RandomAccessFile	seek + tamaños deterministas
Lectura UTF autocontenida	writeUTF/readUTF	Guarda longitud
Campos de longitud fija exacta	writeChars + StringBuffer	Se garantiza tamaño constante

---

PUNTOS CRÍTICOS DEL PROFESOR

- **Cada char = 2 bytes**, siempre.
  Esto afecta a todos los cálculos de tamaño.
- **Acceso aleatorio real** → solo posible si el tamaño del registro es fijo.
- El orden de lectura debe ser **idéntico** al de escritura.
  Si escribes `int` y lees `double`, rompes el puntero.
- `RandomAccessFile` NO es adecuado para texto puro → usa DataInputStream/DataOutputStream si quieres más control.
- `FileInputStream` y `FileOutputStream` solo sirven para flujos secuenciales; intentar hacer acceso aleatorio con ellos es un error conceptual.

4️⃣ Fijar formato del registro de cadenas de caracteres

En un fichero binario con **acceso aleatorio**, cada registro debe ocupar **exactamente el mismo número de bytes**.

Esto permite que:

- Cada registro esté perfectamente alineado.
- Puedas saltar de uno a otro con `seek(n * tamaño_registro)`.
- No haya variaciones que “desplacen” los datos y rompan la estructura.

El problema surge con las **cadenas de texto**, ya que:

- “Ana” ocupa 3 caracteres → 6 bytes
- “Luis” ocupa 4 caracteres → 8 bytes
- “Guillermo” ocupa 9 caracteres → 18 bytes
- Etc.

Esto hace **imposible** calcular posiciones fijas en el archivo.

Por eso, **las cadenas deben ocupar siempre el mismo tamaño**, aunque el nombre real sea más corto.

¿Cómo se soluciona?

Fijando una longitud exacta, por ejemplo:

- Nombre: **10 caracteres** → 20 bytes
- Apellido: **15 caracteres** → 30 bytes

Y:

- Si la palabra es más corta → se **rellena**.
- Si es más larga → se **recorta**.

Existen dos formas de conseguirlo:
StringBuffer + setLength() o **==String.format()**.==

Método 1: StringBuffer + setLength()

Este método “clásico” garantiza que la cadena tenga exactamente `n` caracteres.

- Si falta longitud → añade `\u0000`
- Si sobra → recorta

Es el método más **antiguo y compatible** con los registros binarios del PDF.

Ejemplo

```
StringBuffer sb = new StringBuffer(nombre);
sb.setLength(10);              // recorta o rellena con '\u0000'
raf.writeChars(sb.toString()); // escribe 10 chars (20 bytes)
```

¿Qué pasa internamente?

Si `nombre = "Ana"`:

```
"A" "n" "a" "\u0000" "\u0000" "\u0000" "\u0000" "\u0000" "\u0000" "\u0000"
```

Se guardan **20 bytes** (2 por carácter).

Lectura correspondiente

```
char[] nombre = new char[10];
for (int i = 0; i < 10; i++) nombre[i] = raf.readChar();
String nombreStr = new String(nombre).trim();
```

`trim()` elimina:

- espacios
- tabulaciones
- saltos de línea
- **caracteres nulos (`\u0000`)**

Resultado:

```
"   Ana\u0000\u0000\u0000\u0000"   →   "Ana"
```

Ventajas de este método

- Perfecto para ficheros de **registros fijos**.
- Control total sobre caracteres reales y de relleno.

Método 2: String.format()

`String.format()` permite fijar:

- **ancho mínimo**
- **alineación**
- **recorte automático**

Es más flexible y legible.

```
String.format(%[flags][width][.precision]conversión)
```

- `%` inicio
- `-` izquierda
- `0` relleno con ceros
- `width` ancho mínimo
- `.precision` máximo
- tipos: `s` , `d` , `f` …

Ejemplo aplicado:

```
String fijo = String.format("%-" + len + "." + len + "s", s);
```

Esto garantiza:

- Alineado a la izquierda (`%-`)
- Relleno con espacios si es más corto
- Recorte si es más largo (`.len`)

Con `len = 6` :

- `"Ana"` → `"Ana "`
- `"Martha"` → `"Martha"`
- `"Demasiado"` → `"Demasi"`

Otros ejemplos

```
String.format("%6s", "Ana");   // "   Ana"
String.format("%-6s", "Ana");  // "Ana   "
String.format("%06d", 42);     // "000042"
```

5️⃣ Escritura con RandomAccessFile (crear Empleados.dat)

```
import java.io.IOException;
import java.io.RandomAccessFile;

public class EmpleadosBinarios {
    public static void main(String[] args) {
        String[] nombres = {"Ana", "Luis", "Marta", "Carlos", "Eva"};
        int[] dept = {10, 20, 10, 30, 10};
        double[] salario = {1500.0, 1800.5, 1600.0, 2100.75, 1700.25};

        try (RandomAccessFile raf = new RandomAccessFile("Empleados.dat", "rw")) {

            for (int i = 0; i < nombres.length; i++) {

                raf.writeInt(i + 1); // ID

                StringBuffer sb = new StringBuffer(nombres[i]);
                sb.setLength(10); // 10 caracteres (20 bytes)
                raf.writeChars(sb.toString());

                raf.writeInt(dept[i]);     // departamento
                raf.writeDouble(salario[i]); // salario
            }

            System.out.println("Fichero binario creado correctamente.");

        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

¿Qué pasa en el bucle?

- ID → 4 bytes
- Nombre → 10 chars → 20 bytes
- Departamento → 4 bytes
- Salario → 8 bytes

Total por registro: **36 bytes**

6️⃣ Lectura con RandomAccessFile (mostrar Empleados.dat)

```java
import java.io.IOException;
import java.io.RandomAccessFile;

public class LeerEmpleadosBinarios {
    public static void main(String[] args) {
        try (RandomAccessFile raf = new RandomAccessFile("Empleados.dat", "r")) {

            int id, dept;
            double salario;
            char[] nombre = new char[10];
            String nombreStr;

            while (raf.getFilePointer() < raf.length()) {

                id = raf.readInt();

                for (int i = 0; i < nombre.length; i++) {
                    nombre[i] = raf.readChar();
                }

                nombreStr = new String(nombre).trim();

                dept = raf.readInt();
                salario = raf.readDouble();

                System.out.printf(
                    "ID: %d | Nombre: %-10s | Dpto: %d | Salario: %.2f%n",
                    id, nombreStr, dept, salario
                );
            }

        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

Notas

- `getFilePointer()` devuelve la posición actual.
- `length()` indica el tamaño total del archivo.
- El orden de lectura debe coincidir con el orden de escritura.
- Si se rompe el orden → el puntero queda desalineado y se lee “basura binaria”.

7️⃣ Ejercicios ejemplo

---

1) Escribir y leer enteros secuencialmente (E11)

```java
import java.io.RandomAccessFile;
import java.io.IOException;

public class E11 {
    public static void main(String[] args) throws IOException {

        try (RandomAccessFile raf = new RandomAccessFile("nums.bin", "rw")) {

            raf.setLength(0);

            for (int i = 1; i <= 5; i++)
                raf.writeInt(i * 10);

            raf.seek(0);

            for (int i = 0; i < 5; i++)
                System.out.println(raf.readInt());
        }
    }
}
```

---

2) Acceso directo a una posición fija (E12)

```java
import java.io.RandomAccessFile;
import java.io.IOException;

public class E12 {
    public static void main(String[] args) throws IOException {

        try (RandomAccessFile raf = new RandomAccessFile("slots.bin", "rw")) {

            raf.setLength(0);

            int[] datos = {100, 200, 300};

            for (int i = 0; i < datos.length; i++) {
                long offset = i * 4L;
                raf.seek(offset);
                raf.writeInt(datos[i]);
            }

            raf.seek(1 * 4L);
            int segundo = raf.readInt();

            System.out.println("Segundo = " + segundo);
        }
    }
}
```

---

3) Cadenas cortas con writeUTF/readUTF (E13)

```java
import java.io.RandomAccessFile;
import java.io.IOException;

public class E13 {
    public static void main(String[] args) throws IOException {

        try (RandomAccessFile raf = new RandomAccessFile("nombres.bin", "rw")) {

            raf.setLength(0);

            raf.writeUTF("Ana");
            raf.writeUTF("Luis");
            raf.writeUTF("Marta");

            raf.seek(0);

            System.out.println(raf.readUTF());
            System.out.println(raf.readUTF());
            System.out.println(raf.readUTF());
        }
    }
}
```

---

4) Registros de tamaño fijo (id + nombre20 + edad) (E14)

```java
import java.io.RandomAccessFile;
import java.io.IOException;

public class E14 {

    static final int NCHARS = 20;
    static final int BYTES_REG = 4 + (2 * NCHARS) + 4;

    public static void main(String[] args) throws IOException {

        try (RandomAccessFile raf = new RandomAccessFile("personas.dat", "rw")) {

            raf.setLength(0);

            raf.seek(0 * BYTES_REG);
            raf.writeInt(1);
            escribirNombreFijo(raf, "Ana", NCHARS);
            raf.writeInt(30);

            raf.seek(1 * BYTES_REG);
            raf.writeInt(2);
            escribirNombreFijo(raf, "Lluis", NCHARS);
            raf.writeInt(25);

            raf.seek(1 * BYTES_REG);
```

```
            int id = raf.readInt();
            String nombre = leerNombreFijo(raf, NCHARS);
            int edad = raf.readInt();

            System.out.println(id + " " + nombre + " " + edad);
        }
    }

    static void escribirNombreFijo(RandomAccessFile raf, String s, int len) throws
IOException {
        if (s == null) s = "";
        if (s.length() > len) s = s.substring(0, len);

        String ajustado = String.format("%-" + len + "s", s);

        for (int i = 0; i < len; i++)
            raf.writeChar(ajustado.charAt(i));
    }

    static String leerNombreFijo(RandomAccessFile raf, int len) throws IOException {
        StringBuilder sb = new StringBuilder(len);

        for (int i = 0; i < len; i++)
            sb.append(raf.readChar());

        return sb.toString().trim();
    }
}
```

Clase 3 – Ejercicios 1 y 2 TAREA 3 OBLIGATORIA

1️⃣ Ejercicio 1 – Escritura y lectura secuencial con RandomAccessFile

En este ejercicio usamos **RandomAccessFile**, pero de manera secuencial, para entender cómo:

- escribe valores binarios,
- lee en el mismo orden,
- y mueve automáticamente el puntero interno.

```
try (Scanner sc = new Scanner(System.in);
     RandomAccessFile raf = new RandomAccessFile("nums.bin", "rw")) {

    raf.setLength(0); // Empezar con el archivo vacío

    // --- Escritura de 5 enteros ---
    for (int i = 1; i <= 5; i++) {

        System.out.print("Introduce tu entero favorito " + i + ": ");

        int num = sc.nextInt(); // valor introducido por teclado
```

```
        // writeInt escribe siempre 4 bytes → el puntero avanza +4
        raf.writeInt(num);
    }

    // Posición final del puntero: 5 * 4 bytes = 20
    System.out.println("getFilePointer = " + raf.getFilePointer());

    raf.seek(0); // volvemos al byte 0 para leerlo todo

    // --- Lectura secuencial ---
    for (int i = 0; i < 5; i++) {
        int n = raf.readInt();  // lee 4 bytes → avanza automáticamente
        System.out.println("[" + i + "] = " + n);
    }
}
```

🔍 Explicación condensada

- `writeInt()` escribe 4 bytes por cada entero.
- Tras escribir 5 enteros → el puntero queda en byte **20**.
- `readInt()` lee 4 bytes y avanza igual que `writeInt()`.
- Aquí no hay offset manual porque estamos **leyendo en orden**.

2️⃣ Ejercicio 2 – Acceso directo a posiciones fijas

Aquí aparece el **primer uso real del acceso aleatorio**:
en vez de leer todo el archivo, **saltamos directamente al segundo entero**.

```
try (Scanner sc = new Scanner(System.in);
     RandomAccessFile raf = new RandomAccessFile("slots.bin", "rw")) {

    raf.setLength(0); // archivo limpio

    // --- Escritura de enteros en "slots" de 4 bytes ---
    for (int i = 0; i < 5; i++) {

        System.out.print("Introduce el entero " + (i + 1) + ": ");
        int n = sc.nextInt();

        // Cada entero ocupa 4 bytes → la posición del slot es i * 4
        long offset = i * 4L;

        raf.seek(offset);   // mover el puntero al slot deseado
        raf.writeInt(n);    // escribir los 4 bytes del entero
    }

    // --- Leer SOLO el segundo valor ---
    raf.seek(1 * 4L);  // segundo entero → byte 4
    int segundo = raf.readInt();
```

```
    System.out.println("Segundo valor = " + segundo);
}
```

🔍 Explicación

- Cada entero ocupa **4 bytes** → podemos calcular la posición exacta.
- `seek(i * 4)` permite saltar directamente al entero número *i* sin leer los anteriores.
- En enteros esto es fácil porque **siempre** ocupan 4 bytes.
- **La importancia real aparece cuando se usan cadenas**:
  - Cada string tiene tamaño distinto.
  - Solo funciona el acceso aleatorio si **fijamos un tamaño fijo**, como se hará en los ejercicios 4+.

🧩 Conclusión

- RandomAccessFile guarda datos en **binario**, tal como están en memoria.
- En el Ejercicio 1, se trabaja secuencialmente para observar el movimiento del puntero.
- En el Ejercicio 2, se demuestra el acceso aleatorio real:
  - calcular offsets,
  - usar `seek()`,
  - escribir y leer en posiciones exactas.
- Estos dos ejercicios preparan el terreno para los registros de **tamaño fijo**, donde la importancia del offset es máxima, sobre todo para **cadenas de texto** que no tienen longitud constante.
